A Separation Kernel Formal Security Policy

David Greve and Matthew Wilding

Rockwell Collins, Inc.
Advanced Technology Center
Cedar Rapids, IA 52498 USA

{dagreve, mmwildin}@rockwellcollins.com

March 30, 2003

1 Introduction

High assurance product evaluation requires precise, unambiguous specifications. Some high assurance products are relied upon to process information containing military or commercial secrets, and it is important to guarantee that no unauthorized interference or eavesdropping can occur. A formal specification of what the system allows and guards against is called a *formal security policy*. The construction of a formal security policy that describes the needed behavior of a security-critical system under evaluation is now commonly required for a high level certification.

A computing system that supports multiple independent levels of security (MILS, a.k.a. MSL or multiple security levels) provides protections to guarantee that information that is assigned different security levels is handled appropriately. The design of MILS/MSL systems guaranteed to perform correctly with respect to security considerations is a daunting challenge. An innovation first published in the early 1980's for architecting secure systems involves the application of a *separation kernel* to reduce the security burden [6]. Interaction between applications is mediated by the separation kernel, which enforces a security policy of information flow and data isolation on those interactions. Architecting a MILS/MSL system using a separation kernel breaks the security challenge into two smaller challenges: (1) building and verifying a dependable separation kernel and (2) building applications that, relying upon protections afforded by the separation kernel,

enforce sensible system security policies.

Broadly speaking, a good specification of a system component has two characteristics. First, it can be mapped to a concrete component implementations using convenient and reliable methods. That is, the specification can be *proved*. Second, it encapsulates needed behavior so that the larger system can benefit from an assurance that the specification holds. That is, the specification can be *used*.

We present in this paper a security property that we believe has both these desired properties. We are currently demonstrating that the security can be proved by showing that it holds of a system under development. The goal of this paper is to show that the security policy can be used, which we demonstrate two ways. First, we introduce some theorems similar to what others have used to describe a separation kernel and prove that our specification implies theirs. Second, we formalize an example application that uses the separation kernel and show that the separation kernel security policy implies that the application works properly.

We use the Common Lisp programming language to describe our security policy [7]. Although widely used as a programming language, Common Lisp is an unusual notation for describing mathematical theorems. We have choosen it for our work because of its usefulness in modeling and reasoning about computing systems [1, 2, 3, 8], especially given the support afforded by the ACL2 theorem proving system for proving theorems expressed in Common Lisp [4]. The definitions and proofs presented in this paper have each been checked using ACL2 version 2.6. In addition, the assumptions we make that are introduced as constraints on the functions of the specifications have been proved using ACL2 to be satisfiable.

Ultimately, we will prove that an implementation adheres to the security policy being presented in this note and use the ACL2 theorem prover to check the proofs. We crafted this policy to provide an unambiguous description for what a separation kernel provides, and constructing such a policy is a step toward the goal of building a formally verified separation kernel implementation, which is our ultimate goal. However, the immediate goal of this paper is to introduce a separation kernel security policy and show that it describes the separation kernel's behavior adequately enough to reason about an example application that relies upon it.

2 Separation Kernel Formal Security Policy

The formal security policy describes abstractly what a separation kernel does. The machine supports a number of *partitions* whose names are provided by the constant function `(allparts)`. We use the notation of ACL2's `encapsulate` command to indicate a function of no arguments that returns a single value.

```
((allparts) => *)
```

One of the partitions is designated the "current" partition. The function `current` calculates the current partition given a machine state.

```
((current *) => *)
```

We use the notation of ACL2's `defthm` command, which presents a theorem expressed in Common Lisp notation, to indicate a property about the functions `current` and `allparts`.

```
(defthm current-is-partition
  (member (current st) (allparts)))
```

Associated with partitions are memory segments. Memory segments have names and are intended to model portions of the machine state. The names of the memory segments associated with a particular partition are available from the function `segs`, which takes as an argument the name of the partition. (Note that since `segs` is a function only of partition name — not, for example, a function of machine state — the assignment of segments to partitions is assumed invariant.)

```
((segs *) => *)
```

The values in a machine state that are associated with a memory segment are extracted by the function `select`. Select takes two arguments: the name of the partition and the machine state.

```
((select * *) => *)
```

The separation kernel enforces a communication policy on the memory segments. This policy is modeled with the function `dia`, which represents the pairs of memory segments for which direct interaction is allowed. The function takes as an argument a memory segment name and returns a list of memory segments that are allowed to effect it. (Note that since `dia` is a function of the memory segment name the formalization here implicitly assumes that the communication policy is invariant.)

```
((dia *) => *)
```

The last function constrained in the security policy is `next`, which models one step of computation of the machine state. The function `next` takes as an argument a machine state and returns a machine state that represents the effect of the single step.

```
((next *) => *)
```

The aforementioned constrained functions are used to construct several additional functions. Function `selectlist` takes a list of segments and returns a list of segment values, function `segslist` takes a list of partitions and returns the list of partition names for all the segments, and `run` takes an initial machine state and number of steps and returns an initial machine state updated by executing the number of steps indicated.

```
(defun selectlist (segs st)
  (if (consp segs)
      (cons
       (select (car segs) st)
       (selectlist (cdr segs) st))
    nil))

(defun segslist (partnamelist)
  (if (consp partnamelist)
      (append
       (segs (car partnamelist))
       (segslist (cdr partnamelist)))
    nil))

(defun run (st n)
  (if (zp n)
      st
    (run (next st) (1- n))))
```

The security policy requires that the effect on an arbitrary memory segment `seg` of the execution of one machine step is a function of the set of memory segments that are both allowed to interact with `seg` and are associated with the current partition.

```
(defthm separation
  (let ((segs (intersection-equal (dia seg) (segs (current st1)))))
    (implies
     (and
      (equal (selectlist segs st1) (selectlist segs st2))
```

```
 (equal (current st1) (current st2))
 (equal (select seg st1) (select seg st2)))
(equal
 (select seg (next st1))
 (select seg (next st2))))))
```

That is the entirety of the separation kernel security policy.

3 Relationship with other formalizations

In this section we present several theorems that hold of any system that meets the security policy of the previous section. We prove these theorems because they have been proposed as good properties for a separation kernel in the literature, in informal discussions we have had, or both. Each of these theorems is a special case of the separation axiom of the security policy of the previous section, as verified using the ACL2 theorem prover.

3.1 Exfiltration

When a partition is the currently-executing partition, a partition's memory segments can only be effected in a way that is consistent with the communication policy. We have formalized this property in the following lemma.

```
(defthm exfiltration
  (implies (and
            (equal (intersection-equal (dia seg)
                   (segs (current st1))) nil)
            (equal (current st1) (current st2))
            (equal (select seg st1) (select seg st2)))
           (equal (select seg (next st1))
                  (select seg (next st2)))))
```

Exfiltration is an instance of the separation axiom of the previous section. It is similar to the "Communication Policy" axiom of [5]. However, there appear to be small differences that between our security policy and that of [5][1].

- The formalization presented in this paper does not preclude changes to the state of a partition that are independent of the operation of the machine. This change allows the introduction of such useful things as free-running counters and the asynchronous arrival of (partition specific) information from external sources.

[1] We rely on the published description of the MASK work. The proofs and formal models associated with this work appear not to have been published.

- The communication policy enforced by the separation kernel allows for a finer level of control, since it is at the memory segment level rather than aggregated at the partition (or, to use the MASK terminology, "cell") level. This allows us to make assertions about specific regions of partition memory, allowing us to define "inbox" regions that are distinct from "read only" program memory.

3.2 Mediation

When a partition executes, the effect on a segment does not depend on anything other than the segment's original value and the values of the current partition.

```
(defthm Mediation
  (implies (and (equal (current st1) (current st2))
                (equal (selectlist (segs (current st1)) st1)
                       (selectlist (segs (current st1)) st2))
                (equal (select seg st1) (select seg st2)))
           (equal (select seg (next st1))
                  (select seg (next st2)))))
```

This theorem is very similar to the second separation constraint of [5], and is an instance of the separation axiom of the previous section.

3.3 Infiltration

When a partition executes, the values of the current partition's memory segments do not depend on other segments that should not effect it.

```
(defthm infiltration
  (implies (and
             (equal (current st1) (current st2))
             (equal (selectlist (segs (current st1)) st1)
                    (selectlist (segs (current st1)) st2))
             (member seg (segs (current st1))))
           (iff (equal (select seg (next st1))
                       (select seg (next st2)))
                t))
```

This too is an instance of the separation axiom of the previous section. (It is an instance of mediation as well.)

4 Formalization of a Firewall Application

In this section we formalize the operation of a firewall application that uses the separation kernel formalized in Section 2 and show that by exploiting the separation kernel's security policy we can show that the firewall works properly. This shows that our formalization of the separation kernel security policy is usable as both a specification for the separation kernel and to support applications.

4.1 Functions `black` and `scrub`

In order to prove that a model of a firewall application works, we introduce functions that we can use to describe how a firewall is supposed to behave. It is not immediately obvious how to formalize the correct operation of a firewall, in part because it is difficult to describe what it means for data not to contain sensitive information. We introduce the notion of "black", which is a predicate on a segment name and a system state. The intended interpretation is that black segments do not contain sensitive information that requires protection. We assume the following properties about `black` and `scrub`.

- **spontaneous-generation** If all segments in a system are black, then after the system progresses one step each segment is black.
- **black-scrub** There exists a function "scrub" that modifies a segment so that it is black.
- **black-function-of-segment** Elements of system state that are not associated with the segment are irrelevant in deciding whether a segment is black.
- **current-scrub** The value of current partition is not changed by scrubbing [2].

These assumptions are formalized using constrained ACL2 functions. Function `black` takes a segment name and a machine state and returns whether the segment contains no sensitive data. Function `scrub` takes a segment name and machine state and returns a new machine state where the segment has been modified so as not to contain sensitive information.

[2] For example, one might assume that the portion of the partitioning kernel state that contains `current` is disjoint from `allsegs`, that it is `black`, and that its `dia` contains only itself.

```
((black * *) => *)

((scrub * *) => *)
```

Two functions are defined using the constrained functions.

```
(defun blacklist (segnames st)
   (if (consp segnames)
       (and
        (black (car segnames) st)
        (blacklist (cdr segnames) st))
     t))

(defun scrublist (segs st)
         (if (consp segs)
             (scrublist (cdr segs) (scrub (car segs) st))
           st))
```

We further constrain these functions by adding the following assumptions about them. We believe that these constraints formalize properties of the functions as we have described them informally above.

```
(defthm scrub-commutative
  (equal
   (scrub seg1 (scrub seg2 st))
   (scrub seg2 (scrub seg1 st))))

(defthm segment-scrub-different
  (implies (not (equal seg1 seg2))
           (equal (select seg1 (scrub seg2 st))
                  (select seg1 st))))

(defthm black-scrub
  (equal
   (black seg1 (scrub seg2 st))
   (or
    (equal seg1 seg2)
    (black seg1 st))))

(defthm current-scrub
  (equal
   (current (scrub seg st))
   (current st)))

(defthm spontaneous-generation
  (implies
   (blacklist (segslist (allparts)) st)
   (black seg (next st))))
```

```
(defthm black-function-of-segment
  (implies
   (equal (select x st1) (select x st2))
   (equal (black x st1) (black x st2))))
```

We have shown that the axioms are consistent using ACL2, but are they reasonable for the properties we wish to formalize? In other words, does it formalize a sensible notion of sensitive information? Consider a computing platform that is supposed to handle data of this type. We could imagine extending it so that the system labeled all data with a "black" bit that identifies whether the byte contains sensitive information. Any operation that produces data would set the result's black bit to the "and" of all the input black bits.

Note that each of these assumptions seems reasonable on this enhanced system. In particular,

- spontaneous-generation holds, since any operation will set black bits if every segment in the system has its black bits set. Note that, this framework could model something like a decryption algorithm. Decryption requires keys or algorithms that would not be considered "black" in this framework, so this axiom would be consistent with such models.
- Black-scrub holds since one can "scrub" a data segment by zeroizing all the data and setting the black bits.
- Black-function-of-segment holds since it is straightforward to tell if a segment is black by checking whether all its black bits are set.

Informally, we believe that our formalization of "black" and "scrub" are reasonable in part because in principle it is possible to implement them by adding the enhancement suggested above. We believe that these assumptions constitute a simple but sensible formalization of concepts useful in describing a firewall.

5 A Proved Firewall

We describe a firewall that uses the separation kernel to implement its own security policy. The firewall is implemented as a partition that is guarded by the separation kernel, and we add assumptions about the configuration

of the system and the behavior of the firewall. We use the notion of "black" introduced in the previous section to describe a firewall security policy, and we prove that it is met using the security policy of the security kernel upon which it depends.

We assume the following about the firewall.

- Besides the kernel partition, there is a partition named **b** and a partition named **f**.

```
(defthm allparts-includes
  (and
   (member 'b (allparts))
   (not (equal 'b (kernel-name)))
   (member 'f (allparts))
   (not (equal 'f (kernel-name)))))
```

- When partition **f** is executing, memory segment "outbox" does not transition from black to non-black. The firewall operation therefore is assumed not to allow non-black information to be placed into memory segment "outbox".

```
(defthm firewall-blackens
  (implies
   (and
    (equal (current st) 'f)
    (black 'outbox st))
   (black 'outbox (next st))))
```

- Exactly one segment in **b** is writable from a memory segment that is associated with a partition that is not **b**. The target memory segment is named "outbox" and the source memory segments are all associated only with partition **f**.

```
(defthm dia-setup
  (implies
   (and
    (member seg1 (dia seg2))
    (member seg2 (segs 'b))
    (member seg1 (segs p))
    (not (equal p 'b)))
   (and
    (equal seg2 'outbox)
    (iff
     (member seg1 (segs p2))
     (equal p2 'f)))))
```

These assumptions have been demonstrated to be consistent with the other assumptions introduced earlier in this paper.[3]

We now state the security policy we desire the firewall to enforce: non-black data will never be present in the memory segments of **b**.

```
(defthm firewall-works
  (implies
   (blacklist (segs 'b) st)
   (blacklist (segs 'b) (run st n))))
```

This theorem has been proved. The proof relies upon the assumptions about the security policy and the assumptions about the operation of the firewall. The proof requires the proof of several sublemmas. All the definitions, assumptions, sublemmas, and the final lemma are processed in seconds using ACL2.

6 Summary

We have introduced a formal security policy for a separation kernel, and argued for its usefulness by comparing it with other theorems and using it to formalize a firewall application and proving that the firewall works.

We are now proving that this security policy holds of a separation kernel implementation currently under development.

References

[1] William R. Bevier, Warren A. Hunt Jr., J Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.

[2] David Greve and Matthew Wilding. Evaluatable, high-assurance microprocessors. In *Second Annual High-Confidence Systems and Software Conference (HCSS02)*. National Security Agency, March 2002. Also `http://hokiepokie.org/docs`.

[3] David Greve, Matthew Wilding, and David Hardin. High-speed, analyzable simulators. In *Computer-Aided Reasoning: ACL2 Case Studies*.

[3] ACL2 does not provide a mechanism to extend encapsulated functions conservatively. The consistency of these lemmas was established by proving these constraints hold of the witness function used to show the consistency of the original axioms, and adding these theorems as axioms.

Kluwer Academic Publishers, 2000. Also `http://hokiepokie.org/-docs`.

[4] M. Kaufmann and J S. Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203 – 213, April 1997.

[5] W. Martin, P. White, F.S. Taylor, and A. Goldberg. Formal construction of the mathematically analyzed separation kernel. In *Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00)*. IEEE Computer Society Press, 2000.

[6] J. Rushby. Design and verification of secure systems. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, volume 15, December 1981.

[7] Guy L. Steele. *Common LISP: The Language*. Digital Press, 1984.

[8] Matthew Wilding, David Greve, and David Hardin. Efficient simulation of formal processor models. *Formal Methods in System Design*, 18(3), May 2001.